**MCA Part II**

**Paper-XIII**

# Topic: Process Synchronisation


## Prepared by: Dr. Kiran Pandey
## (School of Computer Science)

## Email-id: kiranpandey.nou@gmail.com


### Introduction

The operating system supports concurrent execution of a program without necessarily supporting elaborate form of memory and file management. This form of operation is also known as multitasking. One of the benefits of multitasking is that several processes can be made to cooperate in order to achieve their goals. To do this, they must do one of the following:

**(i)** **Communicate:** Inter-process communication (IPC) involves sending information from one process to another. This can be achieved using a "mailbox" system, a socket which behaves like a virtual communication network (loopback), or through the use of "pipes". Pipes are a system construction which enable one process to open another process as if it were a file for writing or reading.

**(ii)** **Share Data:** A segment of memory must be available to both the processes. (Most memory is locked to a single process).

**(iii) Waiting:** Some processes wait for other processes to give a signal before continuing. This is an issue of synchronization.

Synchronization is often necessary when processes communicate. Processes are executed with unpredictable speeds. Yet to communicate one process must perform some action such as setting the value of a variable or sending a message that the other detects. This only works if the events perform an action or detect an action are constrained to happen in that order. Thus one can view synchronization as a set of constraints on the ordering of events. The programmer employs a synchronization mechanism to delay execution of a process in order to satisfy such constraints.

## Inter-process Communication and Synchronization

**Inter-process communication** (**IPC**) is a set of programming interfaces that allow a programmer to coordinate activities among different program processes that can run concurrently in an operating system. This allows a program to handle many user requests at the same time.

Processes execute to accomplish specified computations. An interesting and innovative way to use a computer system is to spread a given computation over several processes. The need for such communicating processes arises in parallel and distributed processing contexts. Often it is possible to partition a computational task into segments which can be distributed amongst several processes. Clearly, these processes would then form a set of communicating processes which cooperate in advancing a solution. In a highly distributed, multi-processor system, these processes may even be resident on different machines. In such a case the communication is supported over a network. We shall study some of the basics to be able to do the following:

- How to spawn (or create) a new process.
- How to assign a task for execution to this newly spawned process.
- A few mechanisms to enable communication amongst processes.

- ❖ Synchronization amongst these processes.

In most cases an IPC package is used to establish inter-process communication. Depending upon the nature of the chosen IPC, the package sets up a data structure in kernel space. These data structures are often persistent. So once the purpose of IPC has been fulfilled, this set-up needs to be deleted (a cleanup operation).

Consider a machine with a single printer running a time-sharing operation system. If a process needs to print its results, it must request that the operating system gives it access to the printer's device driver. At this point, the operating system must decide whether to grant this request, depending upon whether the printer is already being used by another process. If it is not, the operating system should grant the request and allow the process to continue; otherwise, the operating system should deny the request and perhaps classify the process as a waiting process until the printer becomes available. Indeed, if two processes were given simultaneous access to the machine's printer, the results would be worthless to both. Consider the following related definitions to understand the example in a better way:

**Critical Resource:** It is a resource shared with constraints on its use (e.g., memory, files, printers, etc).

**Critical Section:** It is code that accesses a critical resource.

**Mutual Exclusion:** At most one process may be executing a critical section with respect to a particular critical resource simultaneously.

In the example given above, the printer is the critical resource. Let's suppose that the processes which are sharing this resource are called process A and process B. The critical sections of process A and process B are the sections of the code which issue the print command. In order to ensure that both processes do not attempt to use the printer at the same, they must be granted mutually exclusive access to the printer driver. First we consider the inter-process communication part. There exist two complementary inter-process communication types:

**a) shared-memory system and**

**b) message-passing system.**

It is clear that these two schemes are not mutually exclusive, and could be used simultaneously within a single operating system.

**Shared-Memory System**

Inter-process communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously. To illustrate the concept of cooperating processes, let's consider the producer-consumer problem, which is a common paradigm for cooperating processes.

A **producer** process produces information that is consumed by a **consumer** process. For example, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader. The producer-consumer problem also provides a useful metaphor for the client-server paradigm. We generally think of a server as a producer and a client as a consumer. For example, a web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource.

One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and

consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used. The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Let's look more closely at how the bounded buffer can be used to enable processes to share memory. The following variables reside in a region of memory shared by the producer and consumer processes:

```
#define BUFFER_SIZE 10

typedef struct {

}item;

item buffer [BUFFER_SIZE] ;

int in = 0 ,-

int out = 0 ;
```

The shared buffer is implemented as a circular array with two logical pointers: in and out. The variable in points to the next free position in the buffer; out points to the first full position in the buffer. The buffer is empty when in == out. the buffer is full when ((in + 1) % BUFFER_SIZE) == out.

The producer process has a local variable nextProduced in which the new item to be produced is stored. The consumer process has a local variable nextConsumed in which the item to be consumed is stored.

This scheme allows at most BUFFER_SIZE - l items in the buffer at the same time. We leave it as an exercise for you to provide a solution where BUFFER-SIZE items can be in the buffer at the same time.

**Message-Passing System**

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. For example, a **chat** program used on the World Wide Web could be designed so that chat participants communicate with one another by exchanging messages.

A message-passing facility provides at least two operations: send(message) and receive (message). Messages sent by a process can be of either fixed or variable size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler. This is a common kind of tradeoff seen throughout operating system design.

If processes *P* and *Q* want to communicate, they must send messages to and receive messages from each other; a **communication link** must exist between them. This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network, but rather with its logical implementation. Here are several methods for logically implementing a link and the send()/receive () operations:

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

We look at issues related to each of these features next.

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication. Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send.0 and receive() primitives are defined as:

- send(P, message)—Send a message to process P.
- receive (Q, message)—Receive a message from process Q.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

This scheme exhibits *symmetry* in addressing; that is, both the sender process and the receiver process must name the other to communicate. A variant of this scheme employs *asymmetry* in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send() and receive () primitives are defined as follows:

- send(P, message)—Send a message to process P.
- receive(id, message)—-Receive a message from any process; the variable *id* is set to the name of the process with which communication has taken place.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions. All references to the old identifier must be found, so that they can be modified to the new identifier. In general, any such hard-coding techniques, where identifiers must be explicitly stated, are less desirable than techniques involving indirection.

With indirect communication, the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. For example, POSIX message queues use an integer value to identify a mailbox. In this scheme, a process can communicate with some other process via a number of different mailboxes. Two processes can communicate only if the processes have a shared mailbox, however. The sendC) and receive () primitives are defined as follows:

- send(A, message)—Send a message to mailbox A.
- receive(A, message)—Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

Now suppose that processes *P|, Vi,* and P3 all share mailbox A Process Pi sends a message to *A,* while both P2 and P3 execute a receive() from A Which process will receive the message sent by Pi? The answer depends on which of the following methods we choose:

- Allow a link to be associated with two processes at most.
- Allow at most one process at a time to execute a receive() operation.
- Allow the system to select arbitrarily which process will receive the message (that is, either P2 or P3, but not both, will receive the message). The system also may define an algorithm for selecting which process will receive the message (that is, *round robin* where processes take turns receiving messages). The system may identify the receiver to the sender.

A mailbox may be owned either by a process or by the operating system. If the mailbox is owned by a process (that is, the mailbox is part of the address space of

the process), then we distinguish between the owner (who can only receive messages through this mailbox) and the user (who can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about who should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

In contrast, a mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following:

- Create a new mailbox.
- Send and receive messages through the mailbox.
- Delete a mailbox.

The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

**Interprocess Synchronization**

When two or more processes work on the same data simultaneously strange things can happen. Suppose, when two parallel threads attempt to update the same variable simultaneously, the result is unpredictable. The value of the variable afterwards depends on which of the two threads was the last one to change the value. This is called a race condition. The value depends on which of the threads wins the race to update the variable. What we need in a multitasking system is a way of making such situations predictable. This is called serialization. Let us study the serialization concept in detail in the next section.

**Serialization**

The key idea in process synchronization is serialization. This means that we have to go to some pains to undo the work we have put into making an operating system perform several tasks in parallel. As we mentioned, in the case of print queues, parallelism is not always appropriate. Synchronization is a large and difficult topic, so we shall only undertake to describe the problem and some of the principles involved here. There are essentially two strategies to serializing processes in a multitasking environment.

- The scheduler can be disabled for a short period of time, to prevent control being given to another process during a critical action like modifying shared data. This method is very inefficient on multiprocessor machines, since all other processors have to be halted every time one wishes to execute a critical section.
- A protocol can be introduced which all programs sharing data must obey. The protocol ensures that processes have to queue up to gain access to shared data. Processes which ignore the protocol ignore it at their own peril (and the peril of the remainder of the system!). This method works on multiprocessor machines also, though it is more difficult to visualize. The responsibility of serializing important operations falls on programmers. The OS cannot impose any restrictions on silly behavior-it can only provide tools and mechanisms to assist the solution of the problem.

**Mutexes: Mutual Exclusion**

When two or more processes must share some object, an arbitration mechanism is needed so that they do not try to use it at the same time. The particular object being shared does not have a great impact on the choice of such mechanisms. Consider the following examples: two processes sharing a printer must take turns using it; if they attempt to use it simultaneously, the output from the two processes may be mixed into an arbitrary jumble which is unlikely to be of any use. Two processes attempting to update the same bank account must take turns; if each process reads the current balance from some database, updates it, and then writes it back, one of the updates will be lost. Both of the above examples can be solved if there is some way for each process to exclude the other from using the shared

object during critical sections of code. Thus the general problem is described as the mutual exclusion problem. The mutual exclusion problem was recognized (and successfully solved) as early as 1963 in the Burroughs AOSP operating system, but the problem is sufficiently difficult widely understood for some time after that. A significant number of attempts to solve the mutual exclusion problem have suffered from two specific problems, the lockout problem, in which a subset of the processes can conspire to indefinitely lock some other process out of a critical section, and the deadlock problem, where two or more processes simultaneously trying to enter a critical section lock each other out.

On a uni-processor system with non-preemptive scheduling, mutual exclusion is easily obtained: the process which needs exclusive use of a resource simply refuses to relinquish the processor until it is done with the resource. A similar solution works on a preemptively scheduled uni-processor: the process which needs exclusive use of a resource disables interrupts to prevent preemption until the resource is no longer needed. These solutions are appropriate and have been widely used for short critical sections, such as those involving updating a shared variable in main memory. On the other hand, these solutions are not appropriate for long critical sections, for example, those which involve input/output. As a result, users are normally forbidden to use these solutions; when they are used, their use is restricted to system code.

Mutual exclusion can be achieved by a system of locks. A mutual exclusion lock is colloquially called a mutex. You can see an example of mutex locking in the multithreaded file reader in the previous section. The idea is for each thread or process to try to obtain locked-access to shared data:

**Get_Mutex(m);**

**// Update shared data**

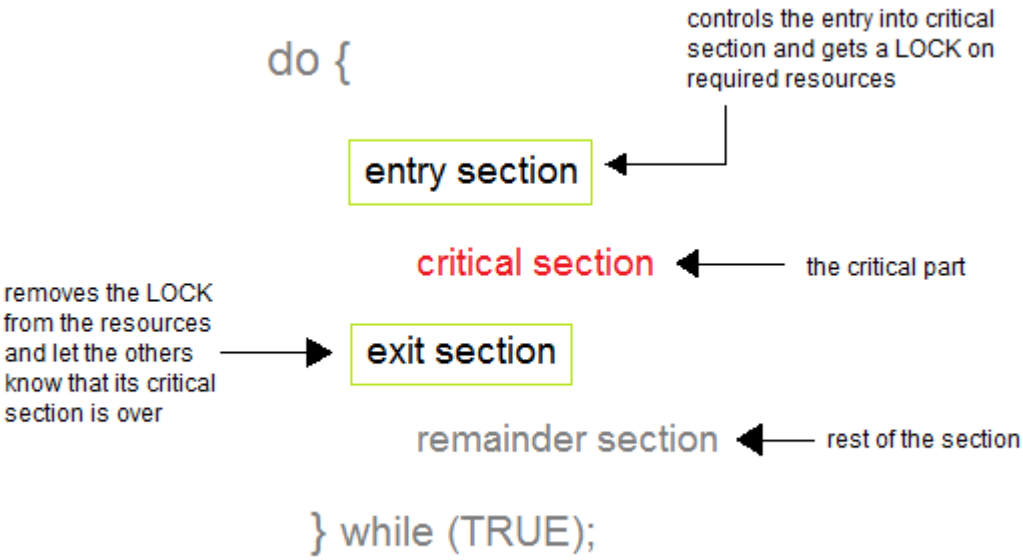
**Release_Mutex(m);**

This protocol is meant to ensure that only one process at a time can get past the function Get_Mutex. All other processes or threads are made to wait at the function Get_Mutex until that one process calls Release_Mutex to release the lock. A method

for implementing this is discussed below. Mutexes are a central part of multithreaded programming.

**Critical Section Problem**

A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes.




**Solution to Critical Section Problem**

A solution to the critical section problem must satisfy the following three conditions:

1. **Mutual Exclusion**

   Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2. **Progress**

If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

3. **Bounded Waiting**

   After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, system must grant the process permission to get into its critical section.

In discussion of the critical section problem, we often assume that each thread is executing the following code. It is also assumed that:

(1) after a thread enters a critical section, it will eventually exit the critical section;

(2) a thread may terminate in the non-critical section. while(true) { entry section critical section exit section non-critical section }

**Critical Sections: The Mutex Solution**

Consider a system consisting of *n* processes {PQ, PI, ..., *P,,~|}.* Each process has a segment of code, called a **critical section,** in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in

its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The *critical-section problem* is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section.** The critical section may be followed by an **exit section.** The remaining code is the **remainder section.**

A critical section is a part of a program in which is it necessary to have exclusive access to shared data. Only one process or a thread may be in a critical section at any one time. The characteristic properties of the code that form a Critical Section are:

- Codes that refer one or more variables in a "read-update-write" fashion while any of those variables is possibly being altered by another thread.
- Codes that alter one or more variables that are possibly being referenced in "read-update-write" fashion by another thread.
- Codes use a data structure while any part of it is possibly being altered by another thread.
- Codes alter any part of a data structure while it possibly in use by another thread.

In the past it was possible to implement this by generalizing the idea of interrupt masks. By switching off interrupts (or more appropriately, by switching off the scheduler) a process can guarantee itself uninterrupted access to shared data. This method has drawbacks:

1. Masking interrupts can be dangerous- there is always the possibility that important interrupts will be missed;

2. It is not general enough in a multiprocessor environment, since interrupts will continue to be serviced by other processors-so all processors would have to be switched off;

3. It is too harsh. We only need to prevent two programs from being in their critical sections simultaneously if they share the same data. Programs A and B might share different data to programs C and D, so why should they wait for C and D?

In 1981 G.L. Peterson discovered a simple algorithm for achieving mutual exclusion between two processes with PID equal to 0 or 1. The code is as follows:

```
 int turn;
 int interested[2];
void Get_Mutex (int pid)

{
int other;
other = 1 - pid;
interested[process] = true;
 turn = pid;
while (turn == pid && interested[other]) // Loop until no one
 {                                       // else is interested
}
}
 Release_Mutex (int pid)
{
interested[pid] = false;
 }
```

Where more processes are involved, some modifications are necessary to this algorithm. The key to serialization here is that, if a second process tries to obtain the mutex, when another already has it, it will get caught in a loop, which does not terminate until the other process has released the mutex. This solution is said to involve busy waiting**-**i.e., the program actively executes an empty loop, wasting CPU cycles, rather than moving the process out of the scheduling queue. This is also called a spin lock, since the system 'spins' on the loop while waiting. Let us see another algorithm which handles critical section problem for **n** processes.

**Dekker's Solution for Mutual Exclusion**

**Dekker's algorithm** is the first known correct solution to the mutual exclusion problem in concurrent programming. The solution is attributed to Dutch mathematician Th. J. Dekker by Edsger W. Dijkstra in an unpublished paper on sequential process descriptions and his manuscript on cooperating sequential processes. It allows two threads to share a single-use resource without conflict, using only shared memory for communication.

It avoids the strict alternation of a naïve turn-taking algorithm, and was one of the first mutual exclusion algorithms to be invented.

If two processes attempt to enter a critical section at the same time, the algorithm will allow only one process in, based on whose turn it is. If one process is already in the critical section, the other process will busy wait for the first process to exit. This is done by the use of two flags, wants_to_enter[0] and wants_to_enter[1], which indicate an intention to enter the critical section on the part of processes 0 and 1, respectively, and a variable turn that indicates who has priority between the two processes.

Dekker's algorithm can be expressed in pseudocode, as follows.[3]

```
variables
    wants_to_enter : array of 2 booleans
    turn : integer
wants_to_enter[0] ← false
wants_to_enter[1] ← false
turn ← 0   // or 1
```

```
p0:
```

```
  wants_to_enter[0] ← true
  while wants_to_enter[1] {
    if turn ≠ 0 {
      wants_to_enter[0] ← false
      while turn ≠ 0 {
        // busy wait
      }
      wants_to_enter[0] ← true
    }
  }
  // critical section
  ...
  turn ← 1
  wants_to_enter[0] ← false
  // remainder section
p1:
  wants_to_enter[1] ← true
  while wants_to_enter[0] {
    if turn ≠ 1 {
      wants_to_enter[1] ← false
```

```
      while turn ≠ 1 {
        // busy wait
      }
      wants_to_enter[1] ← true
    }
  }
  // critical section
  ...
  turn ← 0
  wants_to_enter[1] ← false
  // remainder section
```

Processes indicate an intention to enter the critical section which is tested by the outer while loop. If the other process has not flagged intent, the critical section can be entered safely irrespective of the current turn. Mutual exclusion will still be guaranteed as neither process can become critical before setting their flag (implying at least one process will enter the while loop). This also guarantees progress as waiting will not occur on a process which has withdrawn intent to become critical. Alternatively, if the other process's variable was set the while loop is entered and the turn variable will establish who is permitted to become critical. Processes without priority will withdraw their intention to enter the critical section until they are given priority again (the inner while loop). Processes with priority will break from the while loop and enter their critical section.

**Bakery'sAlgorithm**

Bakery algorithm handles critical section problem for n processes as follows:

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P i and P j receive the same number, if i < j , then P i is served first; else P j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...
- Notation <= lexicographical order (ticket #, process id #) o (a,b) < (c,d) if a < c or if a = c and b < d o max(a0, . . . , an-1) is a number, k , such that k >= ai for i = 0, . . . , n – 1
- Shared data **Boolean** choosing[n]; //initialise all to false **int** number[n]; //initialise all to 0
- Data structures are initialized to false and 0, respectively.

The algorithm is as follows:

```
do { choosing[i] = true;

number[i] = max(number[0], number[1], ...,number[n-1]) + 1;

choosing[i] = false;

for(int j = 0; j < n; j++)

{

 while (choosing[j]== true)

 {

/*do nothing*/

}

while ((number[j]!=0) && (number[j],j)< (number[i],i))

// see Reference point
```

```
{
 /*do nothing*/
}
}
 do critical section number[i] = 0;
 do remainder section
}
while (true)
```

**Questions**

1. What are race conditions? How race conditions occur in operating system?
2. What is mutual exclusion? Explain.
3. Define critical section.
4. Discuss various schemes of interprocess communication.
5. Explain Dekker's solution for mutual exclusion.